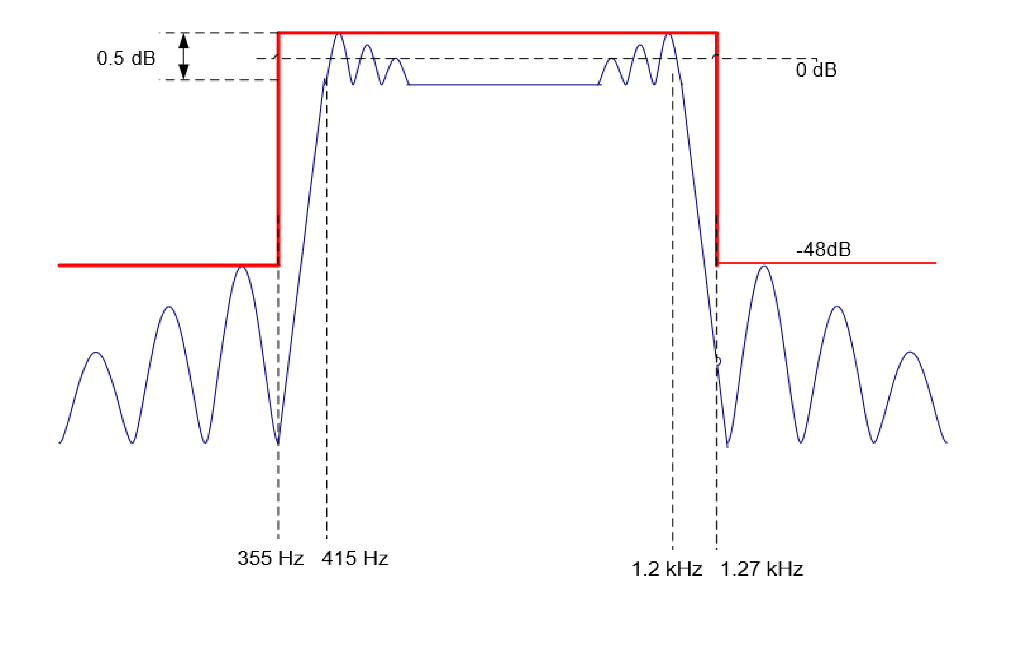
Correct design of the filter in Matlab with proof of the expected frequency response[ 4 ]

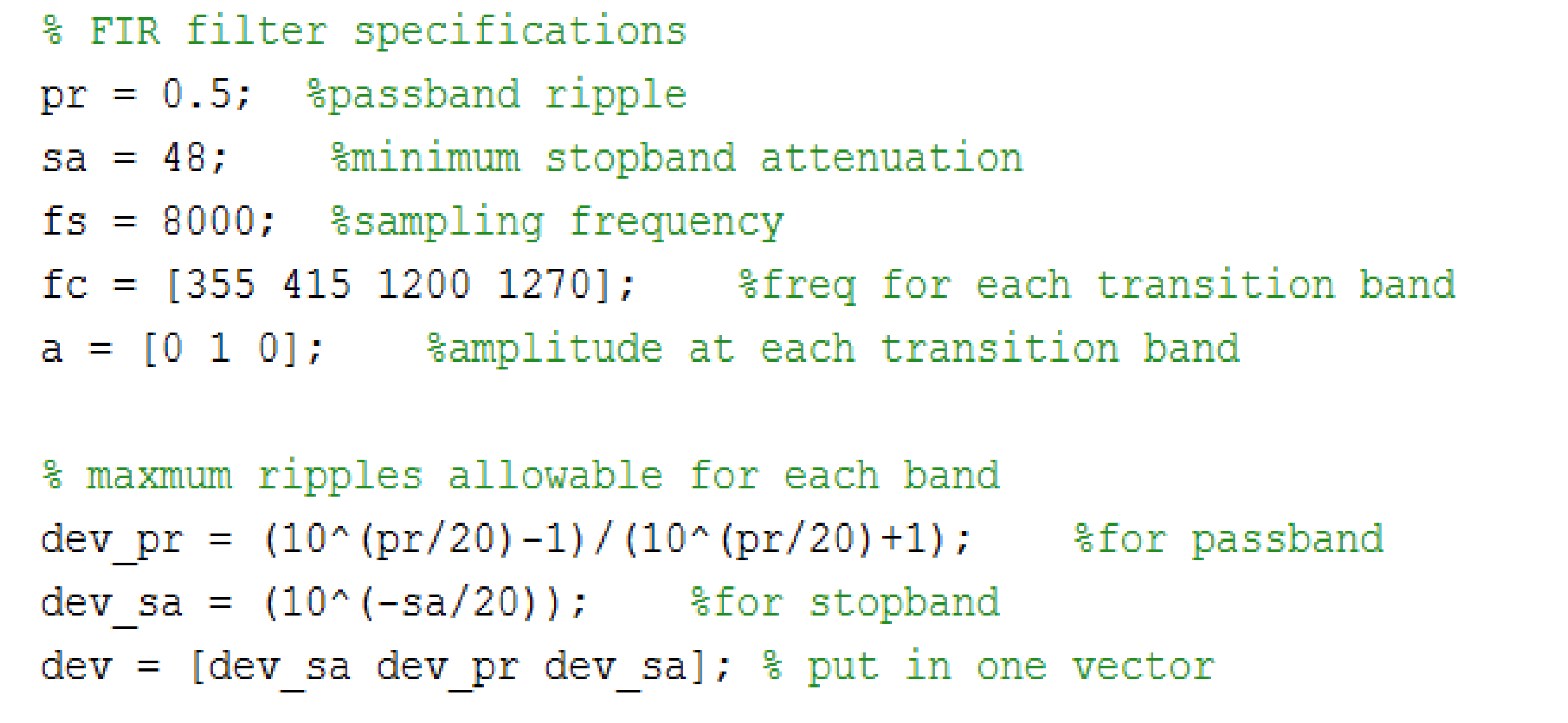
Design a FIR filter using MATLAB that fulfils the specifications given below:



where

|  |  |  |
| --- | --- | --- |
| Passband ripple | 0.5dB |  |
| Min. stopband attenuation | 48dB |  |
| Sampling frequency | 8k Hz |  |
| Passband cut-off frequency | 415 Hz | 1.2 kHz |
| Stopband cut-off frequency | 355 Hz | 1.27 kHz |

Set up parameters required for the FIR filter in MATLAB:

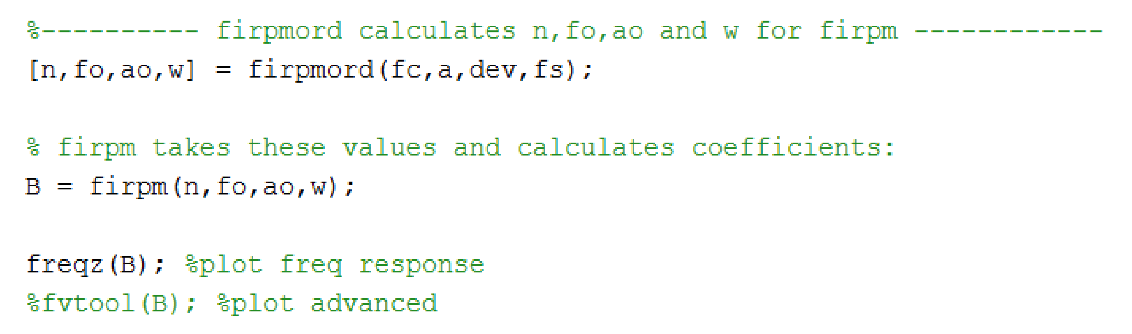


In MATLAB two functions: *firpmord, firpm\** are used to implement the filter specified above. (\*: Section 5 – FIR Filters and their Practical Design)

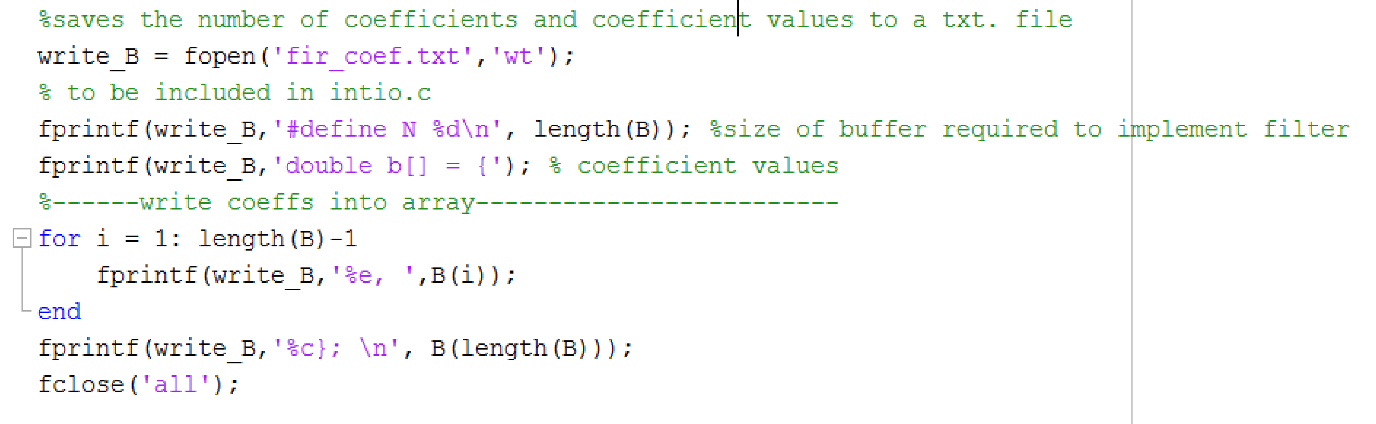
These two functions calculate coefficients of FIR filters using Parks-McClellan algorithm. Zeros are placed in the z-plane to ensure that we achieve an equiripple design method. i.e. Constant pass-band ripples and constant stopband ripples.

MATLAB code is set up based on the model given in the Section 5 lecture notes: *firmord* takes in four parameters: cut-off frequencies (as a vector), band amplitude (as a vector), maximum deviation for each band and sampling frequency to calculate another four parameters (N, Fo, Ao and W) \*\* which are then used by the function *firpm.* *firpm* takes in these values to calculate filter coefficients which will approximately meet the design specification.

*(N - approx. filter order; Fo – normalised frequency band edges; Ao – frequency band amplitude; W - weights) \*\**

**

We then look at the frequency response with B factor to see if it needs to add 1 or 2 to meet the specifications. After we are satisfied with the frequency response we obtained we then write the size and values into a text file which will be included into intio.c file later.



Using MATLAB function *freqz* to plot the frequency response of our filter: in pass-band filter is linear phase. In order to put cursors at all four cur-off frequencies at the same time, we use the MATLAB function *fvtool.*



On both graphs the x-axis is normalised to Nyquist frequency thus need to multiply 4000 (half the sampling frequency) when converting back to frequency in Hz.

The cut-off frequencies (in ascending order) shown on the graph below are *0.08874512\*4000 =* ***354.98Hz,*** *0.1033936\*4000 =* ***413.57Hz,*** *0.2993164\*4000 =* ***1197.27Hz,*** *and 0.3175049\*4000 =* ***1270.02Hz*** respectively which approximately meet the specification given at the beginning of the report.

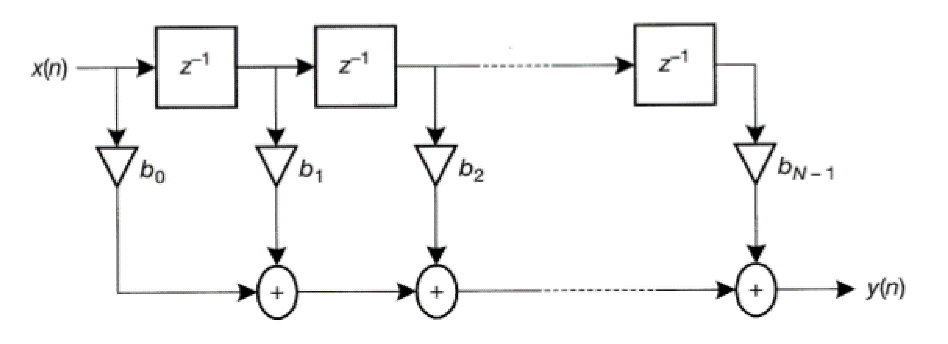


Implementation of a basic non - circular FIR filter in C with explanation of your code. Include benchmark data at different compiler optimization levels.[6 ]

The transfer function of a digital filter of Mth order is given by:

𝐻(𝑧) = 𝑏0 + 𝑏1𝑧−1 + ⋯+ 𝑏𝑀𝑧−𝑀

This can be thought of as an implement of a delay pipeline (shown below):

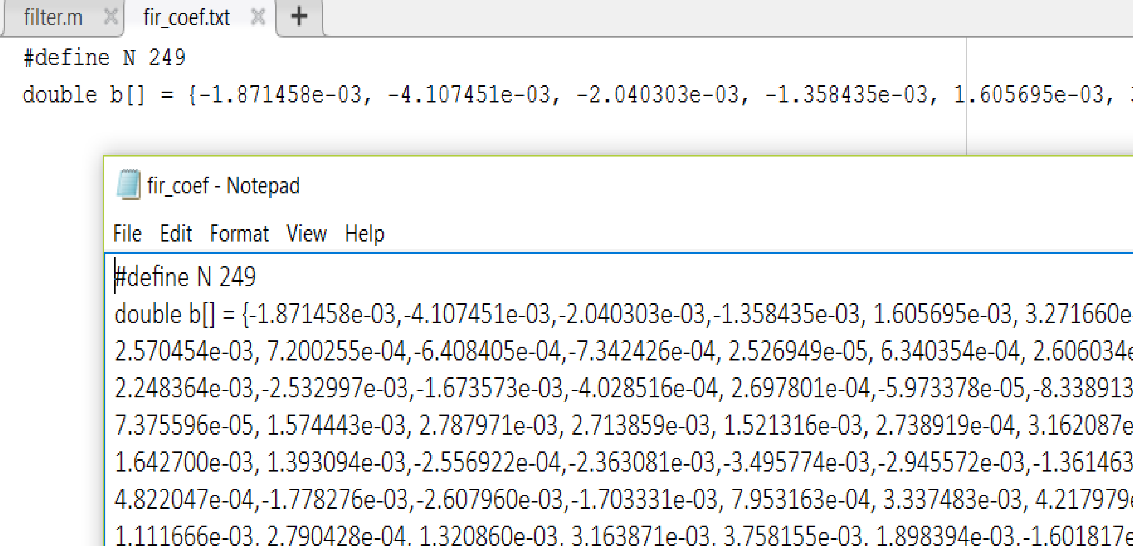


To implement this in c we need to store our input samples in a buffer, apply a multiply accumulate algorithm to the samples and the coefficients of the filter (i.e. convolution), shift the buffer each time a new sample comes in and store the new sample at the beginning of the buffer. By doing this way we discard the oldest one sample at the end and only take care of the latest N samples.

A multiply accumulate algorithm or equivalently a convolution sum has a form:

In which *b(k), the coefficients,* are already calculated using MATLAB and only need to be included in .c file. N as shown in the file snippet below will be our buffer size.

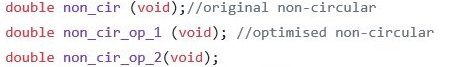




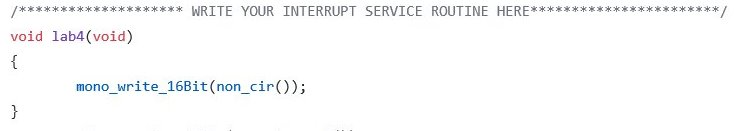
Originally, we make use of the entire length of b and apply the convolution sum one by one. Buffer size is defined and buffer is initialised to zero in global declaration section.



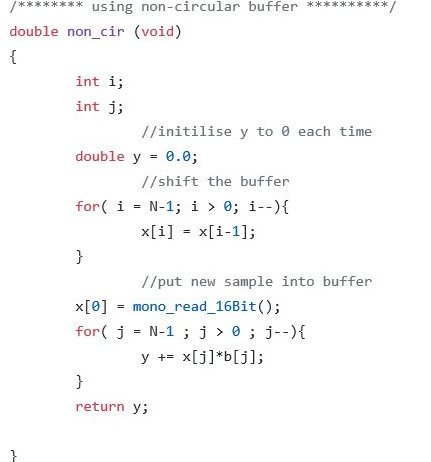
We have three attempts when optimising non-circular methods



In ISR we call the non-circular buffer function and write it to the output:

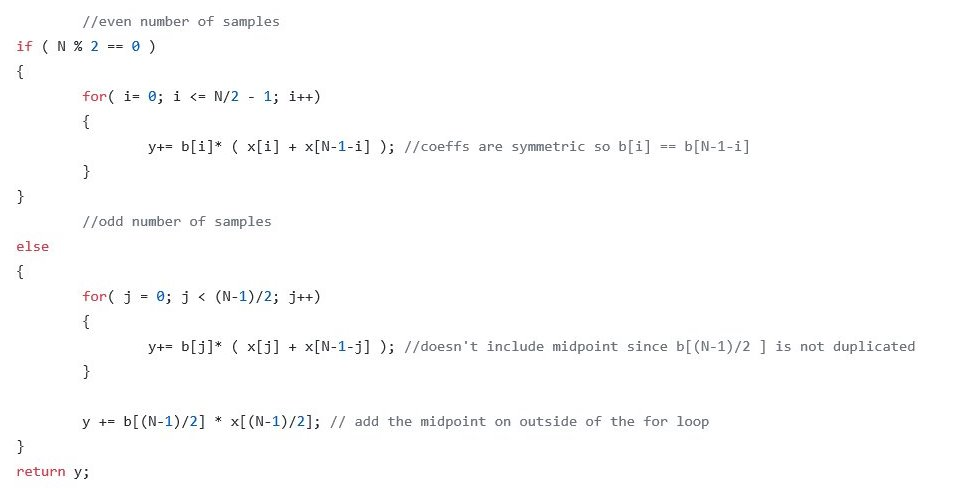


A variable y is defined to store the result of multiply accumulate, each time the value of the MAC will be added on top of y and y is returned after algorithm finishes. We initialise y to zero each time the function is called. The samples are delayed/shifted using a for loop with N iteration steps. After the buffer is shifted we have a space to store a new data from Codec at the right-hand side of the buffer. Then another *for* loop is used to accumulate adds.

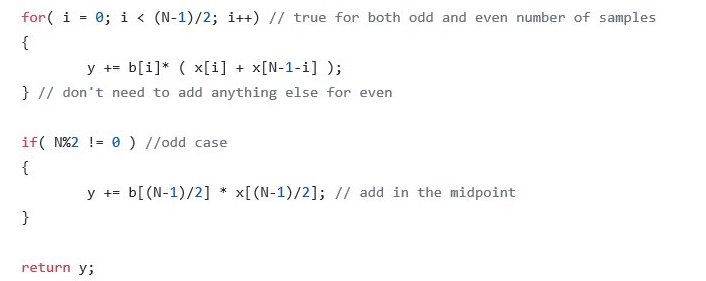


Then we realise the filter coefficients *b(k)* are real and symmetric, so instead of using all the values in *b* array we use first half the array and multiply each of them with the sum of mirror imaged index of *x*(samples).

By doing so we save instruction cycles (and memory…?maybe) for using only half of the coefficients. Our first try is to differentiate between even and odd cases and implement two if statements for them respectively (which basically follow the same flow). Method used to classify the cases is using *modulus, %2.* In even cases, the modulus of filter length will result in 0. Whereas, the odd length will return 1.



Then we optimise it with only one if statement to check if the size is even or odd. Before the midpoint of *b* is reached we do the convolution sum regardless of the even or odd case. After all the elements before midpoint have gone through the accumulate we check if the buffer size is even or odd. If it is even we just return y value we have already obtained, if it is odd we multiply the midpoint of buffer *b* and *x*, and add it on top of y.



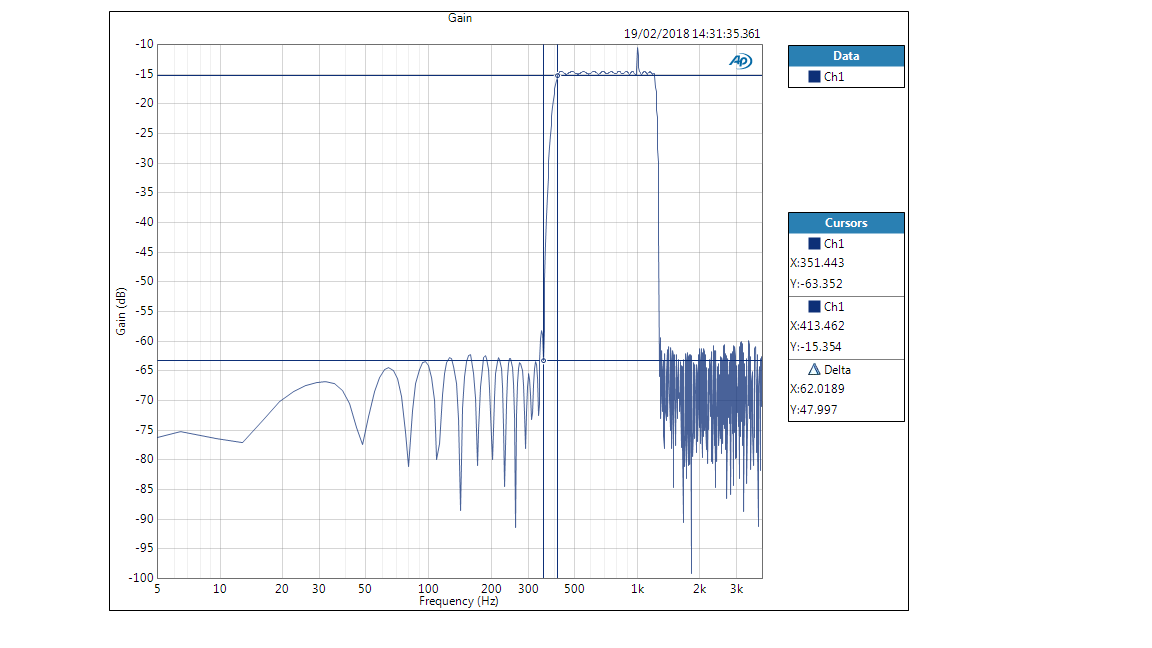
Compare the three attempts with different compiler optimisation levels:

|  |  |  |  |
| --- | --- | --- | --- |
| function | No optimisation | -o0 | -o2 |
| Non\_cir | 15,089 | 12,337 | 3842 |
| Non\_cir\_op\_1 | 12,595 | 9335 | 1268 |
| Non\_cir\_op\_2 | 12,589 | 9335 | 1153 |

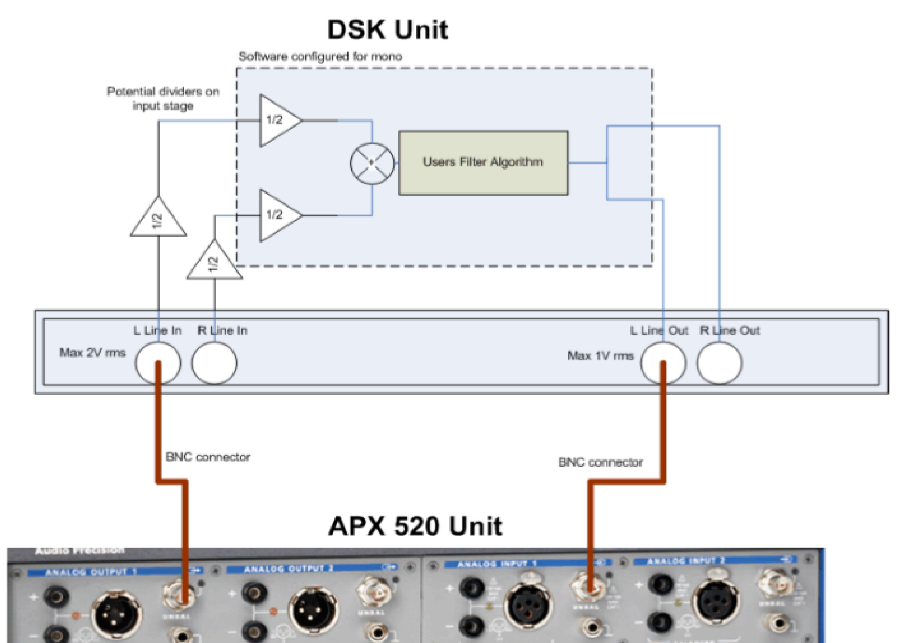
Show the frequency response for your fastest implementation in C from network analyser.Is your filter linear phase? Is the gain what you expect?[ 5 ]

To check the performance of our code we use Audio Precision apx in the lab to generate Bode plots.

Our fastest code is implemented using double sized buffer when circular buffer method.



(…BAD GRAPH!!!…. Passband ripple needs to be checked …..see if within 0.5Db; offset of gain—check center of passband; check maximum ripple in stopband)



As for the gain issue the image included in appendix of lab booklet shows that there are 2 triangular blocks each with factor of ½ which combines to be ¼. Then convert it to gain in dB: 20log(1/4) = -12.04dB. as we can see this will give us a gain offset of 12Db. The maximum ripple in stopband is defined ad -48dB, -48-12 = -60dB which means the maximum ripple we can see in the graph should not exceed this boundary. This is roughly achieved since there are two sidelobes next to the passband that have a gain higher than -60dB.

On lower right of the graph we can see that the minimum ripple in passband is -15.354Db………………………………………………………….